



10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要良心教育、做专业教育，更要做受人尊敬的职业教育。

《从实践中学嵌入式 LINUX 操作系统》

作者：华清远见

专业始于专注 卓识源于远见

第 3 章 嵌入式 Linux 编程环境

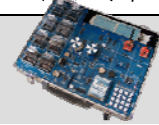
本章简介

本章内容包括常用的 Linux 开发工具使用技巧和 Linux 编程技术，重点介绍常用的 Linux 编程工具和技巧。通过本章学习可以使读者快速掌握基本的 Linux 开发工具，为后续的嵌入式 Linux 开发打下基础。

.....

3.1

Linux 编辑器 vi 的使用



Linux 系统提供了一个完整的编辑器家族系列，如 ed、ex、vi 和 emacs 等。按功能可以分为两大类：行编辑器（ed、ex）和全屏编辑器（vi、emacs）。行编辑器每次只能对单行进行操作，使用起来很不方便，而全屏编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，从而克服了行编辑的那种不直观的操作方式，便于用户学习和使用，具有强大的功能。

vi 是 Linux 系统的第一个全屏交互式编辑程序，它从诞生至今一直得到广大用户的青睐，历经数十年仍然是人们主要使用的文本编辑工具，足以其生命力之强，而强大的生命力是其强大的功能带来的。由于大多数读者在此之前都已经用惯了 Windows 的 Word 等编辑器，因此，在刚刚接触时总会或多或少不适应，但只要习惯之后，就能感受到它的方便与快捷。

3.1.1 vi 的模式

vi 有 3 种模式，分别为命令行模式、插入模式及底行模式，下面具体介绍各模式的功能。

1. 命令行模式

用户在用 vi 编辑文件时，最初进入的为一般模式。在该模式中可以通过上下移动光标进行“删除字符”或“整行删除”等操作，也可以进行“复制”、“粘贴”等操作，但无法编辑文字。

2. 插入模式

只有在该模式下，用户才能进行文字编辑输入，用户可按 Esc 键回到命令行模式。

3. 底行模式

在该模式下，光标位于屏幕的底行。用户可以进行文件保存或退出操作，也可以设置编辑环境，如寻找字符串、列出行号等。

3.1.2 vi 的基本流程

(1) 进入 vi，即在命令行下输入“vi hello（文件名）”，此时进入的是命令行模式，光标位于屏幕的上方，如图 3.1 所示。

(2) 在命令行模式下输入“i”进入插入模式，如图 3.2 所示。可以看出，在屏幕底部显示有“插入”表示插入模式，在该模式下可以输入文字信息。

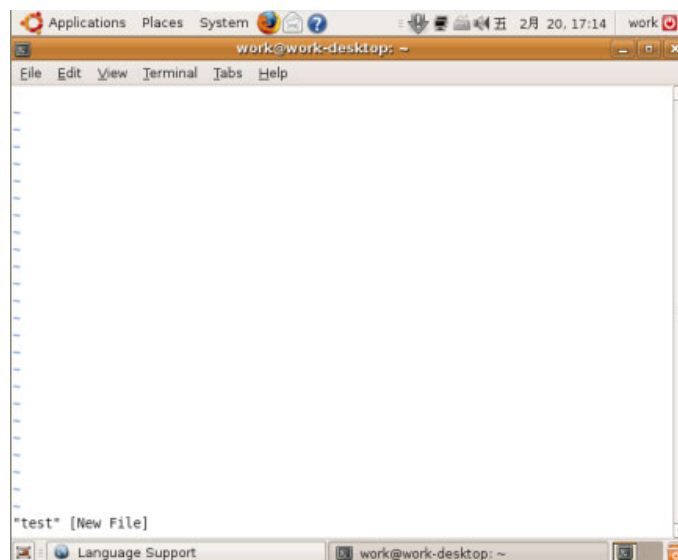


图 3.1 进入 vi 命令行模式

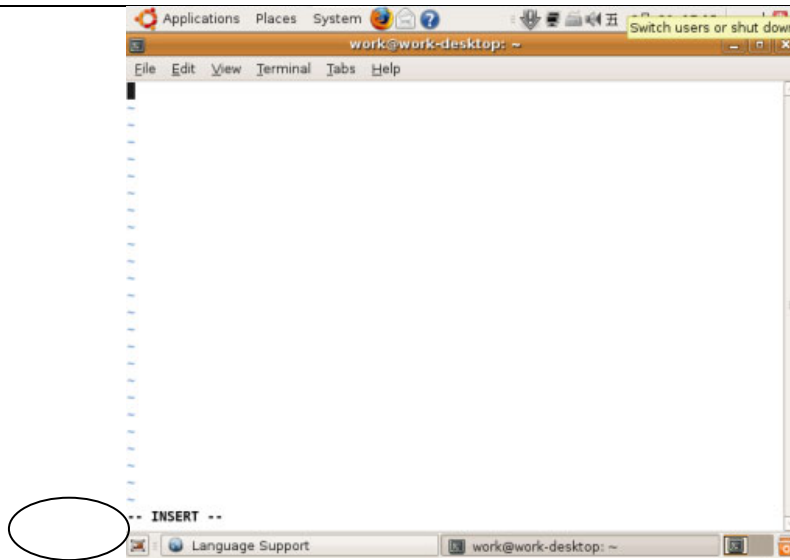


图 3.2 进入 vi 插入模式

(3) 最后，在插入模式中按 Esc 键，则当前模式转入命令行模式，并在底行行中输入“:wq”（存盘退出）进入底行模式，如图 3.3 所示。

这样，就完成了简单的 vi 操作流程：命令行模式→插入模式→底行模式。由于 vi 在不同的模式下有不同的操作功能，因此，读者一定要时刻注意屏幕最下方的提示，分清所在的模式。

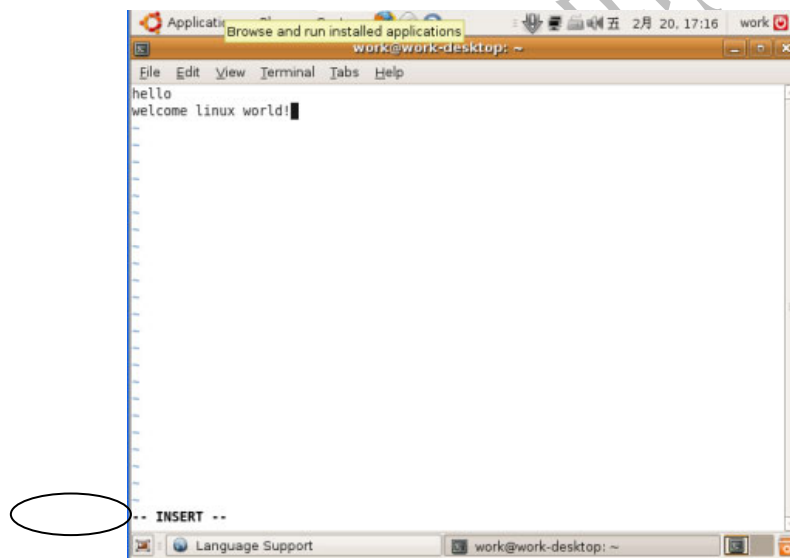


图 3.3 进入 vi 底行模式

3.1.3 vi 的各模式功能键

(1) 命令行模式常见功能键如表 3.1 所示。

表 3.1 vi 命令行模式功能键

目 录	目 录 内 容
I	切换到插入模式，此时光标位于开始输入文件处
A	切换到插入模式，并从目前光标所在位置的下一个位置开始输入文字
O	切换到插入模式，且从行首开始插入新的一行
Ctrl+B	屏幕往“后”翻动一页
Ctrl+F	屏幕往“前”翻动一页
Ctrl+U	屏幕往“后”翻动半页

Ctrl+D	屏幕往“前”翻动半页
0 (数字 0)	光标移动到本行的开头
G	光标移动到文章的最后
nG	光标移动到第 n 行
\$	移动到光标所在行的“行尾”
n<Enter>	光标向下移动 n 行
/name	在光标之后查找一个名为 name 的字符串
?name	在光标之前查找一个名为 name 的字符串
X	删除光标所在位置的“后面”一个字符

续表

目 录	目 录 内 容
X	删除光标所在位置的“前面”一个字符
dd	删除光标所在行
ndd	从光标所在行开始向下删除 n 行
yy	复制光标所在行
nyy	复制光标所在行开始的向下 n 行
p	将缓冲区内的字符粘贴到光标所在位置 (与 yy 搭配)
U	恢复前一个动作

(2) 插入模式的功能键只有一个，也就是 Esc (退出到命令行模式)。

(3) 底行模式常见功能键如表 3.2 所示。

表 3.2 vi 底行模式功能键

目 录	目 录 内 容
:w	将编辑的文件保存到磁盘中
:q	退出 vi (系统对做过修改的文件会给出提示)
:q!	强制退出 vi (对修改过的文件不做保存)
:wq	存盘后退出
:w [filename]	另存一个名为 filename 的文件
:set nu	显示行号，设定之后，会在每一行的前面显示对应行号
:set nonu	取消行号显示

3.2 gcc 编译器

GNU CC (简称 gcc) 是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++ 和 Objective C 等语言编写的程序。gcc 不仅功能强大，而且可以编译如 C、C++、Objective C、Java、Fortran、Pascal、Modula-3 和 Ada 等多种语言；而且 gcc 又是一个交叉平台编译器，它能够在当前 CPU 平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式领域的开发编译。本章中的示例除非特别注明，否则均采用 gcc 版本为 4.0.0。

如表 3.3 所示是 gcc 支持编译源文件的扩展名及其解释。

表 3.3 gcc 所支持扩展名解释

扩展名	所对应的语言	扩展名	所对应的语言
.c	C 原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序	.h	预处理文件（头文件）
.m	Objective-C 原始程序	.o	目标文件
.i	已经过预处理的 C 原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的 C++原始程序		

3.2.1 gcc 编译流程解析

如本章开头提到的，gcc 的编译流程分为 4 个步骤，分别为：

- (1) 预处理（Pre-Processing）。
- (2) 编译（Compiling）。
- (3) 汇编（Assembling）。
- (4) 链接（Linking）。

下面就具体来查看一下 gcc 是如何完成这 4 个步骤的。

首先，有以下 hello.c 源代码：

```
#include<stdio.h>
int main()
{
    printf("Hello! This is our embedded world!\n");
    return 0;
}
```

1. 预处理阶段

在该阶段，编译器将上述代码中的 `stdio.h` 编译进来，并且用户可以使用 gcc 的选项“-E”进行查看，该选项的作用是让 gcc 在预处理结束后停止编译过程。

```
[root@localhost Gcc]# gcc -E hello.c -o hello.i
```

在此处，选项“-o”是指目标文件；由表 3.3 可知，“.i”文件为已经过预处理的 C 原始程序。以下列出了 hello.i 文件的部分内容：

```
typedef int (*__gconv_trans_fct) (struct __gconv_step *,
    struct __gconv_step_data *, void *,
    __const unsigned char *,
    __const unsigned char **,
    __const unsigned char *, unsigned char **,
    size_t *);
...
# 2 "hello.c" 2
int main()
{
    printf("Hello! This is our embedded world!\n");
    return 0;
}
```

由此可见，gcc 确实进行了预处理，它把“stdio.h”的内容插入到 hello.i 文件中。

2. 编译阶段

接下来进行的是编译阶段，在这个阶段中，gcc 首先要检查代码的规范性、是否有语法错误等，以确定代码实际要做的工作，在检查无误后，gcc 把代码翻译成汇编语言。用户可以使用“-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

```
[root@localhost Gcc]# gcc -S hello.i -o hello.s
```

下面列出了 `hello.s` 的内容，可见 `gcc` 已经将其转化为汇编语言了，感兴趣的读者可以分析一下这一行简单的 C 语言程序是如何用汇编代码实现的。

```
.file      "hello.c"
.section  .rodata
.align 4
.LC0:
.string   "Hello! This is our embedded world!"
.text
.globl  main
.type   main, @function
main:
    pushl %ebp
    movl  %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    addl $15, %eax
    addl $15, %eax
    shrl $4, %eax
    sall $4, %eax
    subl %eax, %esp
    subl $12, %esp
    pushl $.LC0
    call puts
    addl $16, %esp
    movl $0, %eax
    leave
    ret
.size   main, .-main
.ident  "GCC: (GNU) 4.0.0 20050519 (Red Hat 4.0.0-8)"
.section .note.GNU-stack,"",@progbits
```

3. 汇编阶段

汇编阶段是把编译阶段生成的“`.s`”文件转成目标文件，读者在此使用选项“`-c`”就可看到汇编代码已转换为“`.o`”的二进制目标代码了，如下：

```
[root@localhost Gcc]# gcc -c hello.s -o hello.o
```

4. 链接阶段

在成功编译之后，就进入了链接阶段。在这里涉及一个重要的概念——函数库。

读者可以重新查看这个小程序，在这个程序中并没有定义 `printf` 的函数实现，且在预编译中包含进的“`stdio.h`”中也只有该函数的声明，而没有定义函数的实现。那么，是在哪里实现“`printf`”函数的呢？最后的答案是：系统把这些函数实现都做到名为 `libc.so.6` 的库文件中去了，在没有特别指定时，`gcc` 会到系统默认的搜索路径 `/usr/lib` 下进行查找，也就是链接到 `libc.so.6` 库函数中去，这样就能实现函数 `printf` 了，而这也就是链接的作用。

函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了，其后缀名一般为“`.a`”。动态库与之相反，在编译、链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件加载库，这样，可以节省系统的开销。动态库一般扩展名为“`.so`”，如前面所述的 `libc.so.6` 就是动态库。`gcc` 在编译时默认使用动态库。

完成了链接之后，`gcc` 就可以生成可执行文件，代码如下：

```
[root@localhost Gcc]# gcc hello.o -o hello
```

运行该可执行文件，出现正确的结果如下：

```
[root@localhost Gcc]# ./hello
Hello! This is our embedded world!
```

3.2.2 gcc 编译选项分析

gcc 有超过 100 个的可用选项，主要包括总体选项、告警和出错选项、优化选项和体系结构相关选项，以下对每一类中最常用的选项进行讲解。

1. 总体选项

gcc 的总体选项如表 3.4 所示，很多在前面的示例中已经有所涉及。

表 3.4 gcc 总体选项列表

扩展名	所对应的语言
-c	只是编译不链接，生成目标文件“.o”
-S	只是编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息
-o file	把输出文件输出到 file 中
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录

续表

扩展名	所对应的语言
-L dir	在库文件的搜索路径列表中添加 dir 目录
-static	链接静态库
-llibrary	链接名为 library 的库文件

对于“-c”、“-E”、“-o”、“-S”选项在 3.2.1 节中已经讲解了其使用方法，在此主要讲解另外两个非常常用的库依赖选项：“-I dir”和“-L dir”。

1) -I dir

正如表 3.4 中所述，“-I dir”选项可以在头文件的搜索路径列表中添加 dir 目录。由于 Linux 中头文件都默认放到了/usr/include/目录下，因此，当用户希望添加放置在其他位置的头文件时，就可以通过“-I dir”选项来指定，这样，gcc 就会到相应的位置查找对应的目录。

例如，在/root/workplace/Gcc 目录下有两个文件：

```

/*hello1.c*/
#include<my.h>
int main()
{
    printf("Hello!!\n");
    return 0;
}
/*my.h*/
#include<stdio.h>
    
```

这样，就可在 gcc 命令行中加入-I 选项：

```
[root@localhost Gcc] gcc hello1.c -I /root/workplace/Gcc/ -o hello1
```

这样，gcc 就能够执行出正确结果。

2) -L dir

选项“-L dir”的功能与“-I dir”类似，能够在库文件的搜索路径列表中添加 dir 目录。例如，有程序 hello_sq.c 需要用到目录/root/workplace/Gcc/lib 下的一个动态库 libsung.so，则只需输入如下命令即可：

```
[root@localhost Gcc] gcc hello_sq.c -L /root/workplace/Gcc/lib -lsung -o hello_sq
```

需要注意的是，“-I dir”和“-L dir”都只是指定了路径，而没有指定文件，因此不能在路径中包含文件名。

另外，值得详细解释一下的是“-l”选项，它指示 gcc 去链接库文件 libsunq.so。由于在 Linux 下的库文件命名时有一个规定：必须以 l、i、b 3 个字母开头，因此，在用“-l”选项指定链接的库文件名时可以省去 l、i、b 3 个字母。也就是说，gcc 在对“-lsunq”进行处理时，会自动去链接名为 libsunq.so 的文件。

2. 告警和出错选项

gcc 的告警和出错选项如表 3.5 所示。

表 3.5 gcc 告警和出错选项列表

选项	含义
-ansi	支持符合 ANSI 标准的 C 程序
-pedantic	允许发出 ANSI C 标准所列的全部警告信息
-pedantic-error	允许发出 ANSI C 标准所列的全部错误信息
-w	关闭所有告警
-Wall	允许发出 gcc 提供的所有有用的报警信息
-werror	把所有的告警信息转换为错误信息，并在告警发生时终止编译过程

下面结合实例对这几个告警和出错选项进行简单的讲解。

如有以下程序段：

```
#include<stdio.h>

void main()
{
    long long tmp = 1;
    printf("This is a bad code!\n");
    return 0;
}
```

这是一个很糟糕的程序，读者可以考虑一下有哪些问题。

1) -ansi

该选项强制 gcc 生成标准语法所要求的告警信息，尽管这还并不能保证所有没有警告的程序都是符合 ANSI C 标准的。运行结果如下：

```
[root@localhost Gcc]# gcc -ansi warning.c -o warning
warning.c: 在函数“main”中:
warning.c:7 警告: 在无返回值的函数中,“return”带返回值
warning.c:4 警告:“main”的返回类型不是“int”
```

可以看出，该选项并没有发现“long long”这个无效数据类型的错误。

2) -pedantic

允许发出 ANSI C 标准所列的全部警告信息，同样也保证所有没有警告的程序都是符合 ANSI C 标准的。其运行结果如下：

```
[root@localhost Gcc]# gcc -pedantic warning.c -o warning
warning.c: 在函数“main”中:
warning.c:5 警告: ISO C90 不支持“long long”
warning.c:7 警告: 在无返回值的函数中,“return”带返回值
warning.c:4 警告:“main”的返回类型不是“int”
```

可以看出，使用该选项查看出了“long long”这个无效数据类型的错误。

3) -Wall

允许发出 gcc 能够提供的所有有用的报警信息。该选项的运行结果如下：

```
[root@localhost Gcc]# gcc -Wall warning.c -o warning
warning.c:4 警告:“main”的返回类型不是“int”
```



```
warning.c: 在函数“main”中:
warning.c:7 警告: 在无返回值的函数中, “return”带返回值
warning.c:5 警告: 未使用的变量“tmp”
```

使用“-Wall”选项找出了未使用的变量 tmp，但它并没有找出无效数据类型的错误。另外，gcc 还可以利用选项对单独的常见错误分别指定警告。有关具体选项的含义，感兴趣的读者可以查看 gcc 手册进行学习。

3. 优化选项

gcc 可以对代码进行优化，它通过编译选项“-On”来控制优化代码的生成，其中 n 是一个代表优化级别的整数。对于不同版本的 gcc 来讲，n 的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从 0 变化到 2 或 3。

不同的优化级别对应不同的优化处理工作。例如，使用优化选项“-O”主要进行线程跳转(Thread Jump) 和延迟退栈(Deferred Stack Pops) 两种优化；使用优化选项“-O2”除了完成所有“-O1”级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等；选项“-O3”则还包括循环展开和其他一些与处理器特性相关的优化工作。

虽然优化选项可以加速代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句也有可能因为循环展开而变得到处都是，所有这些对调试来讲都将是一场噩梦。所以，笔者建议在调试时最好不要使用任何优化选项，只有当程序在最终发行时才考虑对其进行优化。

4. 体系结构相关选项

gcc 的体系结构相关选项如表 3.6 所示。

表 3.6 gcc 体系结构相关选项列表

选项	含义
-mcpu=type	针对不同的 CPU 使用相应的 CPU 指令，可选择的 type 有 i386、i486、pentium 及 i686 等
-mieee-fp	使用 IEEE 标准进行浮点数的比较
-mno-ieee-fp	不使用 IEEE 标准进行浮点数的比较
-msoft-float	输出包含浮点库调用的目标代码
-mshort	把 int 类型作为 16 位处理，相当于 short int
-mrtld	强行将函数参数个数固定的函数用 ret NUM 返回，节省调用函数的一条指令

这些体系结构相关选项在嵌入式设计中会有较多的应用，读者需根据不同体系结构将对应的选项进行组合处理。在本书后面涉及具体实例会有针对性的讲解。



3.3 gdb 调试器

调试是所有程序员都会面临的问题，如何提高程序员的调试效率，更好、更快地定位程序中的问题，从而加快程序开发的进度，是大家共同面对的。就如读者熟知的 Windows 下的一些调试工具，如 VC 自带的设置断点、单步跟踪等，都受到了广大用户的赞赏。那么，在 Linux 下有什么很好的调试工具呢？

本文所介绍的 gdb 调试器是一款 GNU 开发组织并发布的 UNIX/Linux 下的程序调试工具。虽然它没有图形化的友好界面，但是它强大的功能也足以与微软的 VC 工具等媲美。下面就请跟随笔者一步步学习 gdb 调试器。

3.3.1 gdb 使用流程

这里给出了一个短小的程序，由此带领读者熟悉一下 gdb 的使用流程，建议读者能够实际动手操作。

首先，打开 Linux 下的编辑器 vi，编辑如下代码（由于为了更好地熟悉 gdb 的操作，笔者在此使用 vi 编辑，希望读者能够参见 3.3 节中对 vi 的介绍，并熟练使用 vi）：

```

/*test.c*/
#include <stdio.h>
int sum(int m);
int main()
{
    int i,n=0;
    sum(50);
    for(i=1; i<=50; i++)
    {
        n += i;
    }
    printf("The sum of 1-50 is %d \n", n );
}

int sum(int m)
{
    int i,n=0;
    for(i=1; i<=m;i++)
        n += i;
    printf("The sum of 1-m is %d\n", n);
}
    
```

在保存退出后首先使用 gcc 对 test.c 进行编译。注意，一定要加上选项-g，这样编译出的可执行代码中才包含调试信息；否则，之后 gdb 无法载入该可执行文件。

```
[root@localhost Gdb]# gcc -g test.c -o test
```

虽然这段程序没有错误，但调试完全正确的程序可以更加了解 gdb 的使用流程。接下来就启动 gdb 进行调试。注意，gdb 进行调试的是可执行文件，而不是如.c 的源代码，因此，需要先通过 gcc 编译生成可执行文件才能用 gdb 进行调试。

```

[root@localhost Gdb]# gdb test
GNU Gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/libthread_db.so.1".
(gdb)
    
```

可以看出，在 gdb 的启动画面中指出了 gdb 的版本号、使用的库文件等信息，接下来就进入了由“(gdb)”开头的命令行界面。

(1) 查看文件。在 gdb 中输入“l”（list）就可以查看所载入的文件，代码如下：

```

(gdb) l
1      #include <stdio.h>
2      int sum(int m);
3      int main()
4      {
5          int i,n=0;
6          sum(50);
7          for(i=1; i<=50; i++)
8              {
9                  n += i;
10             }
(gdb) l
11     printf("The sum of 1~50 is %d \n", n );
12
13     }
14     int sum(int m)
15     {
    
```

```

16         int i,n=0;
17         for(i=1; i<=m;i++)
18             n += i;
19         printf("The sum of 1~m is = %d\n", n);
20     }
    
```

可以看出，gdb 列出的源代码中明确地给出了对应的行号，这样就可以大大地方便代码的定位。

(2) 设置断点。设置断点是调试程序中一个非常重要的手段，它可以使程序到一定位置暂停它的运行。因此，程序员在该位置处可以方便地查看变量的值、堆栈情况等，从而找出代码的症结所在。

在 gdb 中设置断点非常简单，只需在“b”后加入对应的行号即可（这是最常用的方式，另外还有其他方式设置断点），代码如下：

```

(gdb) b 6
Breakpoint 1 at 0x804846d: file test.c, line 6.
    
```

要注意的是，在 gdb 中利用行号设置断点是指代码运行到对应行之前将其停止，如上例中，代码运行到第 5 行之前暂停（并没有运行第 5 行）。

(3) 查看断点情况。在设置完断点之后，用户可以输入“info b”来查看设置断点情况，在 gdb 中可以设置多个断点。

```

(gdb) info b
Num Type          Disp Enb Address      What
1 breakpoint      keep y  0x0804846d in main at test.c:6
    
```

(4) 运行代码。接下来就可运行代码了，gdb 默认从首行开始运行代码，输入“r”（run）即可（若想从程序中指定行开始运行，可在 r 后面加上行号）。

```

(gdb) r
Starting program: /root/workplace/Gdb/test
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0x5fb000

Breakpoint 1, main () at test.c:6
6          sum(50);
    
```

可以看到，程序运行到断点处就停止了。

(5) 查看变量值。在程序停止运行之后，程序员所要做的工作是查看断点处的相关变量值。在 gdb 中只需输入“p+变量值”即可，代码如下：

```

(gdb) p n
$1 = 0
(gdb) p i
$2 = 134518440
    
```

在此处，为什么变量“i”的值为如此奇怪的一个数字呢？原因就在于程序是在断点设置的对应行之前停止的，那么在此时并没有把“i”的数值赋为零，而只是一个随机的数字。但变量“n”是在第 4 行赋值的，故在此时已经为零。

(6) 单步运行。单步运行可以使用命令“n”（next）或“s”（step），它们之间的区别在于：若有函数调用的时候，“s”会进入该函数而“n”不会进入该函数。因此，“s”就类似于 VC 等工具中的“step in”，“n”类似于 VC 等工具中的“step over”。它们的使用如下：

```

(gdb) n
The sum of 1-m is 1275
7         for(i=1; i<=50; i++)
(gdb) s
sum (m=50) at test.c:16
16        int i,n=0;
    
```

可见，使用“n”后，程序显示函数 sum 的运行结果并向下执行，而使用“s”后则进入到 sum 函数之中单步运行。

(7) 恢复程序运行。在查看完所需变量及堆栈情况后，就可以使用命令“c”(continue)恢复程序的正常运行。这时，它会把剩余还未执行的程序执行完，并显示剩余程序中的执行结果。以下是之前使用“n”命令恢复后的执行结果：

```
(gdb) c
Continuing.
The sum of 1-50 is :1275

Program exited with code 031.
```

可以看出，程序在运行完后退出，之后程序处于“停止状态”。

3.3.2 gdb 基本命令

`gdb` 的命令可以通过查看帮助进行查找。由于 `gdb` 的命令很多，因此 `gdb` 的帮助将其分成了很多种类(class)，用户可以通过进一步查看相关 class 找到相应命令，如下：

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
...
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

上述列出了 `gdb` 各个分类的命令，注意底部的加粗部分说明其为分类命令。接下来可以具体查找各分类的命令，如下：

```
(gdb) help data
Examining data.

List of commands:

call -- Call a function in the program
delete display -- Cancel some expressions to be displayed when program stops
delete mem -- Delete memory region
disable display -- Disable some expressions to be displayed when program stops
...
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

至此，若用户想要查找 `call` 命令，就可输入“help call”。

```
(gdb) help call
Call a function in the program.
The argument is the function name and arguments, in the notation of the
current working language. The result is printed and saved in the value
history, if it is not void.
```

当然，若用户已知命令名，直接输入“help [command]”也是可以的。

`gdb` 中的命令主要分为以下几类：工作环境相关命令、设置断点与恢复命令、源代码查看命令、查看运行数据相关命令及修改运行参数命令，下面就分别对这几类命令进行讲解。

1. 工作环境相关命令

`gdb` 中不仅可以调试所运行的程序，而且还可以对程序相关的工作环境进行相应的设定，甚至还可以使用 Shell 中的命令进行相关的操作，其功能极其强大。如表 3.7 所示为 `gdb` 常见工作环境相关命令。

表 3.7 `gdb` 工作环境相关命令

命令格式	含义
------	----

set args	指定运行时的参数，如 set args 2
show args	查看设置好的运行参数
path dir	设定程序的运行路径
show paths	查看程序的运行路径
set environment var [=value]	设置环境变量
show environment [var]	查看环境变量
cd dir	进入到 dir 目录，相当于 Shell 中的 cd 命令
pwd	显示当前工作目录
shell command	运行 Shell 的 command 命令

2. 设置断点与恢复命令

gdb 中设置断点与恢复的常见命令如表 3.8 所示。

表 3.8 gdb 设置断点与恢复相关命令

命令格式	含义
info b	查看所设断点
break 行号或函数名 <条件表达式>	设置断点
tbreak 行号或函数名 <条件表达式>	设置临时断点，到达后被自动删除
delete [断点号]	删除指定断点，其断点号为“info b”中的第一栏。若默认断点号则删除所有断点
disable [断点号]	停止指定断点，使用“info b”仍能查看此断点。同 delete 一样，默认断点号则停止所有断点
enable [断点号]	激活指定断点，即激活被 disable 停止的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号] <num>	在程序执行中，忽略对应断点 num 次

续表

命令格式	含义
step	单步恢复程序运行，且进入函数调用
next	单步恢复程序运行，但不进入函数调用
finish	运行程序，直到当前函数完成返回
c	继续执行函数，直到函数结束或遇到新的断点

由于设置断点在 gdb 的调试中非常重要，所以在此着重讲解一下 gdb 中设置断点的方法。

gdb 中设置断点有多种方式：其一是按行设置断点，设置方法在 3.3.1 节已经指出，在此就不重复了；另外，还可以设置函数断点和条件断点，在此结合 3.3.1 节中的代码，具体介绍后两种设置断点的方法。

1) 函数断点

gdb 中按函数设置断点只需把函数名列在命令“b”之后，代码如下：

```
(gdb) b sum
Breakpoint 1 at 0x80484ba: file test.c, line 16.
(gdb) info b
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x080484ba in sum at test.c:16
```

要注意的是，此时的断点实际是在函数的定义处，也就是在 16 行处（注意，第 16 行还未执行）。

2) 条件断点

gdb 中设置条件断点的格式为：

b 行数

或

函数名 if 表达式

具体实例如下:

```
(gdb) b 8 if i==10
Breakpoint 1 at 0x804848c: file test.c, line 8.
(gdb) info b
Num Type          Disp Enb Address      What
1  breakpoint      keep y  0x0804848c in main at test.c:8
    stop only if i == 10
(gdb) r
Starting program: /home/yul/test
The sum of 1-m is 1275

Breakpoint 1, main () at test.c:9
9      n += i;
(gdb) p i
$1 = 10
```

可以看到, 该例中在第 8 行 (也就是运行完第 7 行的 for 循环) 设置了一个 “i==10” 的条件断点, 在程序运行之后可以看出, 程序确实在 i 为 10 时暂停运行。

3. gdb 中源代码查看相关命令

在 gdb 中可以查看源代码以方便其他操作, 其常见的相关命令如表 3.9 所示。

表 3.9 gdb 源代码查看相关命令

命令格式	含义
list <行号><函数名>	查看指定位置代码
file [文件名]	加载指定文件
forward-search 正则表达式	源代码前向搜索
reverse-search 正则表达式	源代码后向搜索
dir dir	停止路径名
show directories	显示定义了了的源文件搜索路径
info line	显示加载到 gdb 内存中的代码

4. gdb 中查看运行数据相关命令

gdb 中查看运行数据是指当程序处于 “运行” 或 “暂停” 状态时, 可以查看的变量及表达式的信息, 其常见命令如表 3.10 所示。

表 3.10 gdb 查看运行数据相关命令

命令格式	含义
print 表达式 变量	查看程序运行时对应表达式和变量的值
x <n/f/u>	查看内存变量内容, 其中 n 为整数表示显示内存的长度, f 表示显示的格式, u 表示从当前地址往后请求显示的字节数
display 表达式	设定在单步运行或其他情况中, 自动显示的对应表达式的内容

5. gdb 中修改运行参数相关命令

gdb 还可以修改运行时的参数, 并使该变量按照用户当前输入的值继续运行。它的设置方法为: 在单步执行的过程中输入命令 “set 变量=设定值”。这样, 在此之后, 程序就会按照该设定的值运行了。下面, 笔者结合 3.3.1 节的代码将 n 的初始值设为 4, 其代码如下:

```
(gdb) b 7
Breakpoint 5 at 0x804847a: file test.c, line 7.
```

```
(gdb) r
Starting program: /home/yul/test
The sum of 1-m is 1275

Breakpoint 5, main () at test.c:7
7          for(i=1; i<=50; i++)
(gdb) set n=4
(gdb) c
Continuing.
The sum of 1-50 is 1279

Program exited with code 031.
```

可以看到，最后的运行结果确实比之前的值大了4。

3.4 make 工程管理器



到目前为止，读者已经了解了如何在 Linux 下使用编辑器编写代码，如何使用 gcc 把代码编译成可执行文件，还学习了如何使用 gdb 来调试程序。那么，所有的工作看似已经完成了，为什么还需要 make 这个工程管理器呢？

所谓工程管理器，顾名思义，是用来管理较多的文件的工具。读者可以试想一下，有一个上百个文件的代码构成的项目，如果其中只有一个或少数几个文件进行了修改，按照之前所学的 gcc 编译工具，就不得不把这所有的文件重新编译一遍，因为编译器并不知道哪些文件是最近更新的，而只知道需要包含这些文件才能把源代码编译成可执行文件，于是，程序员就不得不再重新输入数目如此庞大的文件名以完成最后的编译工作。

人们就希望有一个工程管理器能够自动识别更新了的文件代码，同时又不需要重复输入冗长的命令行，这样，make 工程管理器也就应运而生了。

实际上，make 工程管理器也就是一个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入 Makefile 文件的内容来执行大量的编译工作，用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有 Linux 下的项目编程均会涉及它，希望读者能够认真学习本节内容。

3.4.1 Makefile 基本结构

Makefile 是 make 读入的唯一配置文件，因此本节的内容实际就是讲述 Makefile 的编写规则。在一个 Makefile 中通常包含如下内容：

- (1) 需要由 make 工具创建的目标体 (target)，通常是目标文件或可执行文件。
- (2) 要创建的目标体所依赖的文件 (dependency_file)。
- (3) 创建每个目标体时需要运行的命令 (command)。

它的格式为：

```
target: dependency_files
      command
```

例如，有两个文件分别为 hello.c 和 hello.h，创建的目标体为 hello.o，执行的命令为 gcc 编译指令：gcc -c hello.c，那么，对应的 Makefile 就可以写为：

```
#The simplest example
hello.o: hello.c hello.h
      gcc -c hello.c -o hello.o
```

接着就可以使用 make 了。使用 make 的格式为：make target，这样 make 就会自动读入 Makefile（也可以是首字母小写 makefile）并执行对应 target 的 command 语句，并会找到相应的依赖文件，如下：

```
[root@localhost makefile]# make hello.o
```

```
gcc -c hello.c -o hello.o
[root@localhost makefile]# ls
hello.c hello.h hello.o Makefile
```

可以看到，Makefile 执行了“hello.o”对应的命令语句，并生成了“hello.o”目标体。

3.4.2 Makefile 变量

上面示例的 Makefile 在实际中是几乎不存在的，因为它过于简单，仅包含两个文件和一个命令，在这种情况下完全没有必要编写 Makefile 而只需在 Shell 中直接输入即可。在实际中使用的 Makefile 往往是包含很多的文件和命令的，这也是 Makefile 产生的原因。下面就给出稍微复杂一些的 Makefile 进行讲解。

```
sunq:kang.o yul.o
Gcc kang.o bar.o -o myprog
kang.o : kang.c kang.h head.h
Gcc -Wall -O -g -c kang.c -o kang.o
yul.o : bar.c head.h
Gcc -Wall -O -g -c yul.c -o yul.o
```

在这个 Makefile 中有 3 个目标体 (target)，分别为 sunq、kang.o 和 yul.o，其中，第一个目标体的依赖文件就是后两个目标体。如果用户使用命令“make sunq”，则 make 管理器就是找到 sunq 目标体开始执行。

这时，make 会自动检查相关文件的时间戳。首先，在检查“kang.o”、“yul.o”和“sunq”3 个文件的时间戳之前，它会向下查找那些把“kang.o”或“yul.o”作为目标文件的时间戳。例如，“kang.o”的依赖文件为“kang.c”、“kang.h”、“head.h”，如果这些文件中任何一个的时间戳比“kang.o”新，则命令“gcc -Wall -O -g -c kang.c -o kang.o”将会执行，从而更新文件“kang.o”。在更新完“kang.o”或“yul.o”之后，make 会检查最初的“kang.o”、“yul.o”和“sunq”3 个文件，只要文件“kang.o”或“yul.o”中的任何文件时间戳比 sunq 新，则第二行命令就会被执行。这样，make 就完成了自动检查时间戳的工作，开始执行编译工作。这也就是 make 工作的基本流程。

接下来，为了进一步简化编辑和维护 Makefile，make 允许在 Makefile 中创建和使用变量。变量是在 Makefile 中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标体、依赖文件、命令及 Makefile 文件中的其他部分。在 Makefile 中的变量定义有两种方式：一种是递归展开方式，另一种是简单方式。

递归展开方式定义的变量是在引用该变量时进行替换的，即如果该变量包含了对其他变量的应用，则在引用该变量时一次性将内嵌的变量全部展开。虽然这种类型的变量能够很好地完成用户的指令，但是它也有严重的缺点，如不能在变量后追加内容（因为语句：CFLAGS = \$(CFLAGS) -O 在变量扩展过程中可能导致无穷循环）等。

为了避免上述问题，简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其他变量的引用，从而消除变量的嵌套引用。

递归展开方式的定义格式为：

```
VAR=var
```

简单扩展方式的定义格式为：

```
VAR:=var
```

make 中的变量均使用格式为：

```
$(VAR)
```

下面给出了上例中用变量替换修改后的 Makefile，这里用 OBJS 代替 kang.o 和 yul.o，用 CC 代替 gcc，用 CFLAGS 代替“-Wall -O -g”。这样，在以后修改时，就可以只修改变量定义，而不需要修改下面的定义实体，从而大大简化了 Makefile 维护的工作量。

经变量替换后的 Makefile 如下：

```
OBJS = kang.o yul.o
CC = gcc
CFLAGS = -Wall -O -g
sunq : $(OBJS)
      $(CC) $(OBJS) -o sunq
```



```
kang.o : kang.c kang.h
    $(CC) $(CFLAGS) -c kang.c -o kang.o
yul.o : yul.c yul.h
    $(CC) $(CFLAGS) -c yul.c -o yul.o
```

可以看到，此处变量是以递归展开方式定义的。

Makefile 中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的 `OBJS` 就是用户自定义变量。自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在 Makefile 都会出现的变量，其中部分有默认值，也就是常见的设定值，当然用户也可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及其编译选项。表 3.11 列出了 Makefile 中常见预定义变量及其部分默认值。

表 3.11 Makefile 中常见预定义变量

变 量	含 义
AR	库文件维护程序的名称，默认值为 ar
AS	汇编程序的名称，默认值为 as
CC	C 编译器的名称，默认值为 cc

续表

变 量	含 义
CPP	C 预编译器的名称，默认值为 \$(CC) -E
CXX	C++ 编译器的名称，默认值为 g++
FC	FORTRAN 编译器的名称，默认值为 f77
RM	文件删除程序的名称，默认值为 rm -f
ARFLAGS	库文件维护程序的选项，无默认值
ASFLAGS	汇编程序的选项，无默认值
CFLAGS	C 编译器的选项，无默认值
CPPFLAGS	C 预编译的选项，无默认值
CXXFLAGS	C++ 编译器的选项，无默认值
FFLAGS	FORTRAN 编译器的选项，无默认值

可以看出，上例中的 `CC` 和 `CFLAGS` 是预定义变量，其中由于 `CC` 没有采用默认值，因此，需要把“`CC=gcc`”明确列出来。

由于常见的 `gcc` 编译语句中通常包含目标文件和依赖文件，而这些文件在 Makefile 文件中目标体的一行已经有所体现，因此，为了进一步简化 Makefile 的编写，引入了自动变量。自动变量通常可以代表编译语句中出现的目标文件和依赖文件等，并且具有本地含义（即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件）。表 3.12 列出了 Makefile 中常见的自动变量。

表 3.12 Makefile 中常见自动变量

变 量	含 义
\$*	不包含扩展名的目标文件名称
\$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
\$<	第一个依赖文件的名称
\$?	所有时间戳比目标文件晚的依赖文件，并以空格分开
\$@	目标文件的完整名称
^	所有不重复的依赖文件，以空格分开
%	如果目标是归档成员，则该变量表示目标的归档成员名称

自动变量的书写比较难记，但是在熟练了之后会非常方便，请读者结合下例中的自动变量改写的 Makefile 进行记忆。

```

OBJJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
suno : $(OBJJS)
        $(CC) $^ -o $@
kang.o : kang.c kang.h
        $(CC) $(CFLAGS) -c $< -o $@
yul.o : yul.c yul.h
        $(CC) $(CFLAGS) -c $< -o $@
    
```

另外，在 Makefile 中还可以使用环境变量。使用环境变量的方法相对比较简单，make 在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量。但是，如果用户在 Makefile 中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

3.4.3 Makefile 规则

Makefile 的规则是 make 进行处理的依据，它包括目标体、依赖文件及其之间的命令语句。一般，Makefile 中的一条语句就是一个规则。在上面的例子中，都显式地指出了 Makefile 中的规则关系，如 “\$(CC) \$(CFLAGS) -c \$< -o \$@”，但为了简化 Makefile 的编写，make 还定义了隐式规则和模式规则，下面分别对其进行讲解。

1. 隐式规则

隐式规则能够告诉 make 怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。make 会自动搜索隐式规则目录来确定如何生成目标文件。如上例就可以写成：

```

OBJJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
suno : $(OBJJS)
        $(CC) $^ -o $@
    
```

为什么可以省略后两句呢？因为 make 的隐式规则指出：所有 “.o” 文件都可自动由 “.c” 文件使用命令 “\$(CC) \$(CPPFLAGS) \$(CFLAGS) -c file.c -o file.o” 生成。这样，“kang.o” 和 “yul.o” 就会分别调用 “\$(CC) \$(CFLAGS) -c kang.c -o kang.o” 和 “\$(CC) \$(CFLAGS) -c yul.c -o yul.o” 生成。

表 3.13 给出了常见的隐式规则目录。

表 3.13 Makefile 中常见隐式规则目录

对应语言扩展名	规 则
C 编译: .c 变为.o	\$(CC) -c \$(CPPFLAGS) \$(CFLAGS)
C++ 编译: .cc 或.C 变为.o	\$(CXX) -c \$(CPPFLAGS) \$(CXXFLAGS)
Pascal 编译: .p 变为.o	\$(PC) -c \$(PFLAGS)
Fortran 编译: .r 变为.o	\$(FC) -c \$(FFLAGS)

2. 模式规则

模式规则是用来定义相同处理规则的多个文件的。它不同于隐式规则，隐式规则仅仅能够用 make 默认的变量来进行操作，而模式规则还能引入用户自定义变量，为多个文件建立相同的规则，从而简化 Makefile 的编写。

模式规则的格式类似于普通规则，这个规则中的相关文件前必须用 “%” 标明。使用模式规则修改后的 Makefile 的编写如下：

```

OBJJS = kang.o yul.o
CC = Gcc
CFLAGS = -Wall -O -g
suno : $(OBJJS)
        $(CC) $^ -o $@
    
```

```
%o : %.c
$(CC) $(CFLAGS) -c $< -o $@
```

3.4.4 make 管理器的使用

使用 make 管理器非常简单，只需在 make 命令的后面输入目标名即可建立指定的目标。如果直接运行 make，则建立 Makefile 中的第一个目标。

此外，make 还有丰富的命令行选项，可以完成各种不同的功能。表 3.14 列出了常用的 make 命令行选项。

表 3.14 make 的命令行选项

命令格式	含义
-C dir	读入指定目录下的 Makefile
-f file	读入当前目录下的 file 文件作为 Makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 Makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐式规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，则打印当前目录名

3.5

使用 autotools



在 3.4 节，读者已经了解到了 make 项目管理器的强大功能。的确，Makefile 可以帮助 make 完成它的使命，但要承认的是，编写 Makefile 确实不是一件轻松的事，尤其对于一个较大的项目而言更是如此。那么，有没有一种轻松的手段生成 Makefile，同时又能让用户享受 make 的优越性呢？本节要讲的 autotools 系列工具正是为此而设的，它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成 Makefile，这无疑广大用户的所希望的。另外，这些工具还可以完成系统配置信息的收集，从而方便地处理各种移植性的问题。也正是基于此，现在 Linux 上的软件开发一般都用 autotools 来制作 Makefile。

3.5.1 autotools 使用流程

autotools 是系列工具，读者首先要确认系统是否装了以下工具（可以使用 which 命令进行查看）：

- aclocal。
- autoscan。
- autoconf。
- autoheader。
- automake。

使用 autotools 主要就是利用各个工具的本脚本文件以生成最后的 Makefile。其总体流程如下：

(1) 使用 aclocal 生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义。

(2) 改写“configure.scan”文件，将其重命名为“configure.in”，并使用 autoconf 文件生成 configure 文件。

接下来，将通过一个简单的 hello.c 例子熟悉 autotools 生成 Makefile 的过程。由于在这一过程中会涉及较多的脚本文件，为了更清楚地了解相互之间的关系，强烈建议读者实际动手操作以体会其整个过程。

1. autoscan

它会在给定目录及其子目录树中检查源文件，若没有给出目录，就在当前目录及其子目录树中进行检查。它会搜索源文件以寻找一般的移植性问题并创建一个文件“configure.scan”，该文件就是接下来 autoconf 要用到的“configure.in”原型，代码如下：

```
[root@localhost automake]# autoscan
autom4te: configure.ac: no such file or directory
autoscan: /usr/bin/autom4te failed with exit status: 1
[root@localhost automake]# ls
autoscan.log  configure.scan  hello.c
```

由上述代码可知，autoscan 首先会尝试读入“configure.ac”（同 configure.in 的配置文件）文件，此时还没有创建该配置文件，于是它会自动生成一个“configure.in”的原型文件“configure.scan”。

2. autoconf

configure.in 是 autoconf 的脚本配置文件，它的原型文件“configure.scan”如下：

```
#                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
#The next one is modified by sunq
#AC_INIT(FULL-PACKAGE-NAME,VERSION,BUG-REPORT-ADDRESS)
AC_INIT(hello,1.0)
# The next one is added by sunq
AM_INIT_AUTOMAKE(hello,1.0)
AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

下面对这个脚本文件进行解释：

(1) 以“#”号开始的行为注释。

(2) AC_PREREQ 宏声明本文件要求的 autoconf 版本，如本例使用的版本是 2.59。

(3) AC_INIT 宏用来定义软件的名称和版本等信息，在本例中省略了 BUG-REPORT-ADDRESS，一般为作者的 E-mail。

(4) AM_INIT_AUTOMAKE 是笔者另加的，它是 automake 所必备的宏，也同前面一样，PACKAGE 是所要产生软件套件的名称，VERSION 是版本编号。

(5) AC_CONFIG_SRCDIR 宏用来侦测所指定的源码文件是否存在，以此来确定源码目录的有效性。在此处为当前目录下的 hello.c。

(6) AC_CONFIG_HEADER 宏用于生成 config.h 文件，以便 autoheader 使用。

(7) AC_CONFIG_FILES 宏用于生成相应的 Makefile 文件。

(8) 中间的注释间可以分别添加用户测试程序、测试函数库、测试头文件等宏定义。

接下来首先运行 aclocal，生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义，如下：

```
[root@localhost automake]# aclocal
```

接着运行 autoconf，生成“configure”可执行文件，如下：

```
[root@localhost automake]# autoconf
[root@localhost automake]# ls
aclocal.m4  autom4te.cache  autoscan.log  configure  configure.in  hello.c
```

3. autoheader

接着使用 `autoheader` 命令，它负责生成 `config.h.in` 文件。该工具通常会从 `acconfig.h` 文件中复制用户附加的符号定义，因为此处没有附加符号定义，所以不需要创建 `acconfig.h` 文件，代码如下：

```
[root@localhost automake]# autoheader
```

4. automake

这是创建 `Makefile` 很重要的一步，`automake` 要用的脚本配置文件是 `Makefile.am`，用户需要自己创建相应的文件。然后，`automake` 工具转换成 `Makefile.in`。在该例中，笔者创建的文件为 `Makefile.am`，代码如下：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS= hello
hello_SOURCES= hello.c
```

下面对该脚本文件的对应项进行解释。

(1) 其中的 `AUTOMAKE_OPTIONS` 为设置 `automake` 的选项。由于 GNU（在第 1 章中已经有所介绍）对自己发布的软件有严格的规范，如必须附带许可证声明文件 `COPYING` 等，否则 `automake` 执行时会报错。`automake` 提供了 3 种软件等级：`foreign`、`gnu` 和 `gnits`，供用户选择，默认等级为 `gnu`。在本例中使用 `foreign` 等级，它只检测必需的文件。

(2) `bin_PROGRAMS` 定义要产生的执行文件名。如果要产生多个执行文件，每个文件名用空格隔开。

(3) `hello_SOURCES` 定义“`hello`”这个执行程序所需要的原始文件。如果“`hello`”这个程序是由多个原始文件所产生的，则必须把它所用到的所有原始文件都列出来，并用空格隔开。例如，若目标体“`hello`”需要“`hello.c`”、“`sunq.c`”、“`hello.h`”3 个依赖文件，则定义 `hello_SOURCES=hello.c sunq.c hello.h`。要注意的是，如果要定义多个执行文件，则对每个执行程序都要定义相应的 `file_SOURCES`。

接下来可以使用 `automake` 对其生成“`configure.in`”文件，在这里使用选项“`--adding-missing`”让 `automake` 自动添加一些必需的本脚本文件，代码如下：

```
[root@localhost automake]# automake --add-missing
configure.in: installing './install-sh'
configure.in: installing './missing'
Makefile.am: installing 'depcomp'
[root@localhost automake]# ls
aclocal.m4      autoscan.log  configure.in  hello.c       Makefile.am  missing
autom4te.cache  configure     depcomp      install-sh    Makefile.in  config.h.in
```

可以看到，在 `automake` 之后就可以生成 `configure.in` 文件。

5. 运行 configure

在这一步中，通过运行自动配置设置文件 `configure`，把 `Makefile.in` 变成了最终的 `Makefile`，如下：

```
[root@localhost automake]# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for Gcc... Gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether Gcc accepts -g... yes
checking for Gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of Gcc... Gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
```

可以看到，在运行 `configure` 时收集了系统的信息，用户可以在 `configure` 命令中对其进行配置。在 `./configure` 中的自定义参数有两种：一种是开关式 (`--enable-XXX` 或 `--disable-XXX`)；另一种是开放式，即后面要填入一串字符 (`--with-XXX=yyyy`) 参数。读者可以自行尝试其使用方法。另外，读者可以查看同一目录下的“`config.log`”文件，以方便调试。

到此为止，`Makefile` 就可以自动生成了。回忆整个步骤，用户不再需要定制不同的规则，而只需要输入简单的文件及目录名即可，这样就大大方便了用户的使用。`autotools` 生成 `Makefile` 的流程图如图 3.4 所示。

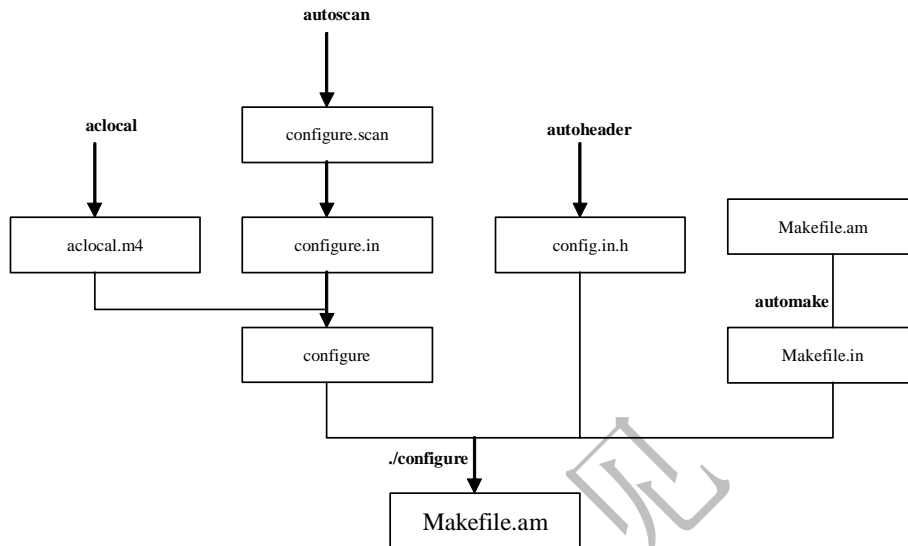


图 3.4 autotools 生成 Makefile 流程图

3.5.2 使用 autotools 生成的 Makefile

`autotools` 生成的 `Makefile` 除具有普通的编译功能外，还具有以下主要功能（感兴趣的读者可以查看这个简单的 `hello.c` 程序的 `Makefile`）。

1. make

输入 `make` 默认执行“`make all`”命令，即目标体为 `all`，其执行情况如下：

```
[root@localhost automake]# make
if gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -DPACKAGE_STRING=\"\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"hello\" -DVERSION=\"1.0\" -I. -I. -g -O2 -MT hello.o -MD -MP -MF
.deps/hello.Tpo -c -o hello.o hello.c; \
then mv -f ".deps/hello.Tpo" ".deps/hello.Po"; else rm -f ".deps/hello.Tpo"; exit 1; fi
gcc -g -O2 -o hello hello.o
```

此时，在本目录下就生成了可执行文件“`hello`”，运行“`./hello`”能出现正常结果，代码如下：

```
[root@localhost automake]# ./hello
Hello!Autoconf!
```

2. make install

此时，会把该程序安装到系统目录中去，代码如下：

```
[root@localhost automake]# make install
if gcc -DPACKAGE_NAME=\"\" -DPACKAGE_TARNAME=\"\" -DPACKAGE_VERSION=\"\" -DPACKAGE_STRING=\"\"
-DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"hello\" -DVERSION=\"1.0\" -I. -I. -g -O2 -MT hello.o -MD -MP -MF
.deps/hello.Tpo -c -o hello.o hello.c; \
then mv -f ".deps/hello.Tpo" ".deps/hello.Po"; else rm -f ".deps/hello.Tpo"; exit 1; fi
gcc -g -O2 -o hello hello.o
```

```
make[1]: Entering directory '/root/workplace/automake'
test -z "/usr/local/bin" || mkdir -p -- "/usr/local/bin"
/usr/bin/install -c 'hello' '/usr/local/bin/hello'
make[1]: Nothing to be done for 'install-data-am'.
make[1]: Leaving directory '/root/workplace/automake'
```

此时，若直接运行 `hello`，也能出现正确结果，代码如下：

```
[root@localhost automake]# hello
Hello!Autoconf!
```

3. make clean

此时，`make` 会清除之前所编译的可执行文件及目标文件（object file, *.o），代码如下：

```
[root@localhost automake]# make clean
test -z "hello" || rm -f hello
rm -f *.o
```

4. make dist

此时，`make` 将程序和相关的文档打包为一个压缩文档以供发布，代码如下：

```
[root@localhost automake]# make dist
[root@localhost automake]# ls hello-1.0-tar.gz
hello-1.0-tar.gz
```

可见该命令生成了一个 `hello-1.0-tar.gz` 的压缩文件。

由上面的讲述读者不难看出，`autotools` 确实是软件维护与发布的必备工具，鉴于此，如今 GUN 的软件一般都是由 `automake` 来制作的。

3.6

本章习题



1. 在 `vi` 中如何进入编辑模式？
2. 在 `vi` 中如何删除整行？
3. `gcc` 的编译流程分为几个步骤？
4. 使用 `gcc` 编译程序中，使用 `-g` 选项的作用是什么？
5. 制作一个简单的 `Makefile` 文件。

联系方式

集团官网：www.hqyj.com

嵌入式学院：www.embedu.org

移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn

物联网学院：www.topsight.cn

研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

华清远见